M&G No. 50002.0007USU1

METHOD AND SYSTEM FOR OPTIMIZING A NETWORK BY INDEPENDENTLY SCALING CONTROL SEGMENTS AND DATA FLOW

Related Application

This application claims the benefit under 35 U.S.C. § 119(e) of U.S. Provisional Application No. 60/191,019, filed March 21, 2000.

Field of the Invention

The present invention relates to a method and apparatus for controlling the flow of data in a network. More specifically, the present invention relates to a network controller that is partitioned into a software controller section and a hardware controller section, the hardware controller section communicating with the software controller section to control the switching of data flow.

Background of the Invention

Local area networks (LANs), which were once merely a desirable technology available to share common network resources, are now an integral part of any information technology (IT) infrastructure. Moreover, the concept of the LAN has expanded to the wide area network (WAN), where remote offices and databases are made available to LAN clients as through they are connected to the same LAN. More recently, virtual private networks (VPN) have been utilized to allow a private intranet to be securely extended across the Internet or other network service, facilitating secure e-commerce and extranet connections with partners, suppliers and customers. The evolution of global networking has rapidly advanced networking topologies.

LAN segments are routinely connected together using a bridge device. The bridge device allowed the two network segments to share traffic despite differences in the network topologies. For Example, a Token Ring network and an Ethernet network can share network traffic using a bridge device.

Routers became popular to couple one LAN to another LAN or WAN. Routers store data packets from one LAN and forward those data packets to another

LAN or WAN. The need for faster communication resulted in the development of the high-speed switch, also referred to as a layer 2/3 switch. High-speed switches move data packets across a network to an end user.

When client-server networks first emerged, servers were generally placed close to their clients. As the applications delivered over networks became more advanced, the servers increased in complexity and capacity. Moreover, applications that ran over these networks such as e-mail, intranet web sites, and Internet gateways, became indispensably pervasive. Supporting these services became critically important, and proved far too complex when servers were widely distributed within the enterprise. As a result, it has become a standard practice to consolidate such resources into server arrays.

A Server array controller is an Internet traffic management device. Server array controllers (hereinafter referred to simply a "controller" or "controllers") control the flow of data packets in and out of an array of application servers. The controller manages and distributes Internet, intranet and other user requests across redundant arrays of network servers, regardless of the platform type. Controllers support a wide variety of network applications such as web browsing, e-mail, telephony, streaming multimedia and other Internet protocol (IP) traffic.

Although advances in data communication technology have dramatically improved the transmission speeds, many problems still exist. Application availability can still be threatened by content failure, software failure or server failure. System resources are often out of balance, with low-performance resources receiving more user requests than high-performance resources being underutilized. Internet Traffic Management (ITM) products are computer systems that sit in the network and process network traffic streams. ITM products switch and otherwise respond to incoming requests by directing them to one of the servers.

A more complete appreciation of the invention and its improvements can be obtained by reference to the accompanying drawings, which are briefly summarized below, to the following detail description of presently preferred embodiments of the invention, and to the appended claims.

Summary of the Invention

In accordance with the invention, an apparatus is provided for directing communications over a network. A control component receives a data flow requesting a resource and determines when the data flow is unassociated with a connection to a requested resource. When the control component determines that the data flow is unassociated with the connection to the requested resource, it will associate a selected connection to the requested resource. A switch component employs the connection associated with the data flow to direct the data flow to the requested resource. The capacity of the switch component and the capacity of the control component are independently scalable to support the number of data flows that are directed to requested resources over the network.

In accordance with other aspects of the invention, the control component employs a buffer to list each data flow that is associated with the connection to the requested resource. The control component can employ a table to list each data flow associated with the connection to the requested resource. Also, the control component may categorize a plurality of data packets for each data flow. Additionally, the control component can determine when an event associated with the data flow occurs and categorize each event.

In accordance with other additional aspects of the invention, a flow signature is associated with the data flow. The flow signature is compared to a set of rules for handling each data flow that is associated with the connection to the requested resource. The flow signature includes information about a source and a destination for each data packet in the data flow. Also, the flow signature can include a timestamp.

In accordance with still other additional aspects of the invention, the switch component collects metrics regarding each connection to each resource. Additionally, a server array controller can act as the control component and the switch component.

In accordance with yet other additional aspects of the invention, the invention provides for directing communications over a network. A flow component receives packets associated with a flow and switches each received packet associated

with the flow to a connection. A control component determines the connection based on information collected by the flow component. The flow segment and the control segment are independently scalable to handle the number of data flows that are directed to requested resources over the network.

In accordance with other additional aspects of the invention, the control component performs control and policy enforcement actions for each flow. Also, the control component collects information regarding each flow including metrics and statistics. The control component performs load balancing for each flow based on the information collected by the flow component. Additionally, a primary control component and a secondary control component can share a load. When the primary control component is inoperative, the secondary control component can take over the actions of the primary control component and the flow component provides the state information for each flow.

In accordance with still other additional aspects of the invention, a server array controller includes the control component and the flow component. Also, the server array controller includes an interface for internal and external networks.

In accordance with yet other additional aspects of the invention, a flow signature is associated with each flow. A timestamp is associated with each flow; and the control component employs the timestamp to determine factors used for load balancing. These factors include most active, least active, time opened and most recent activity. Also, a session that is associated with the flow can include the TCP and UDP protocols. Additionally, the control component can determine when a new flow occurs based on the detection of an event.

In accordance with other additional aspects of the invention, a method is provided for directing communications over a network, including (a) employing a control component to receive a data flow requesting a resource; (b) determining when the data flow is unassociated with a connection to a requested resource; (c) When the data flow is unassociated with the connection to the requested resource, associating a selected connection with the requested resource; and (d) employing the connection associated with the data flow to switch the data flow to the requested resource.

Additionally, the switching capacity and the control capacity are independently scalable to support the number of data flows that are directed to requested resources over the network.

In accordance with yet other additional aspects of the invention, sending state information as multicast messages and other information as point cast messages. Also, responding to messages that are authenticated. Additionally, employing a state sharing message bus (SSMB) between a switch and a control component. The SSMB can be layered on top of a session that may include the TCP and UDP protocols. Further, asynchronous and independent communication may occur between the control component and the switch.

In accordance with still other additional aspects of the invention, associating a flow signature with each flow. Also, comparing when the data flow is associated with the connection to the requested resource and when the interface component determines that the data flow is unassociated with the connection to the requested resource. The comparison is employed to determine the data flow's association with the connection to the requested resource.

The present invention may be implemented as a computer process, a computing system or as an article of manufacture such as a computer program product or computer readable media. The computer program product may be a computer storage media readable by a computer system and encoding a computer program of instructions for executing a computer process. The computer program product may also be a propagated signal on a carrier readable by a computing system and encoding a computer program of instructions for executing a computer process.

These and various other features as well as advantages, which characterize the present invention, will be apparent from a reading of the following detailed description and a review of the associated drawings.

Brief Description of the Drawings

FIGURE 1 is a system diagram of an exemplary partitioned controller;

FIGURE 2 is a system diagram of an exemplary partitioned controller including connections to a client and server;

FIGURE 3 is an exemplary diagram of packet flows from client to server in a partitioned controller;

FIGURE 4 is a table of mnemonics used in an exemplary partitioned controller;

FIGURE5 is a chart showing message fields used in an exemplary partitioned controller;

FIGURE 6 is a table showing message contents used in an exemplary partitioned controller;

FIGURE 7 is a chart showing Boolean variables used in an exemplary partitioned controller;

FIGURE 8 is a chart showing error codes used in an exemplary partitioned controller;

FIGURE 9 is a diagram of formatted message packets using the message fields and variables shown in FIGURES 5 and 6;

FIGURE 10 is another table of mnemonics used in an exemplary partitioned controller;

FIGURE 11 is another chart showing message fields used in an exemplary partitioned controller;

FIGURE 12 is another table showing message contents used in an exemplary partitioned controller;

FIGURE 13 is a chart describing server classes in an exemplary partitioned controller;

FIGURE 14 is a diagram of formatted message packets using the message fields and variables shown in FIGURES 10 and 11;

FIGURE 15 is another chart of Boolean variables used in an exemplary partitioned controller;

FIGURE 16 is another chart of error codes used in an exemplary partitioned controller;

FIGURE 17 is a flow chart of basic operations for a data flow segment (DFS) in an exemplary partitioned controller;

FIGURE 18 is a flow chart of flow activity management for a data flow segment (DFS) in an exemplary partitioned controller;

FIGURE 19 is a flow chart of flow activity management for a control segment (CS) in an exemplary partitioned controller;

FIGURE 20 is a flow chart of message processing for a data flow segment (DFS) in an exemplary partitioned controller;

FIGURE 21 is a flow chart of new flow management for a data flow segment (DFS) in an exemplary partitioned controller; and

FIGURE 22 is a flow chart of message processing for a control segment (CS) in an exemplary partitioned controller in accordance with the present invention.

Detailed Description of the Preferred Embodiment

An embodiment of the invention relates to segmenting part of a server array controller into hardware-optimized and software-optimized portions. The server array controller (controller) performs load balancing and other traffic control functions. An illustration of a server array controller that is segmented in accordance with the present invention is shown in FIGURE 1.

The server array controller (hereinafter referred to simply as a "controller") shown in FIGURE 1 includes one or more network interfaces, and performs the operations of routing, translating, and switching packets. Although FIGURE 1 includes an internal and external network connection, a single network connection is also within the scope of the present invention. The controller maintains the state of each flow of packets. The controller dynamically selects operations on

"flows" based on the content of the packets in the flow. A flow is a sequence of packets that have the same flow signature.

A flow signature is a tuple including information about the source and destination of the packets. In one example, the flow signature is a sextuple of source IP address, source port number, destination IP address, destination port number, protocol, and type-of-service packets in a flow. A flow exists for a finite period. Subsequent flows may have the same flow signature as a previously received flow.

In accordance with invention, the controller includes a Data Flow Segment (DFS) and at least one Control Segment (CS). The DFS includes the hardware-optimized portion of the controller, while the CS includes the software-optimized portions. The DFS performs most of the repetitive chores including statistics gathering and per-packet policy enforcement (e.g. packet switching). The DFS may also perform tasks such as that of a router, a switch, or a routing switch. The CS determines the translation to be performed on each flow of packets, and thus performs high-level control functions and per-flow policy enforcement. Network address translation (NAT) is performed by the combined operation of the CS and DFS.

Although the server array controller is shown as two partitions, it is understood and appreciated that the segmented blocks may be incorporated into one or more separate blocks including, but not limited to, two segments in the same chassis, each CS is a module that plugs into the DFS chassis, and the two segments are merely functional blocks in the same server array controller. The CS and DFS are independently scalable. In one example, multiple DFSs cooperate with a single CS. In another example, multiple CSs cooperate with a single DFS. Additionally, it is envisioned that the functionality of either the DFS or the CS may be separately implemented in software and/or hardware.

The DFS includes at least one connection to a network that can be external or internal. An external network connection is typically to the client-side of the network. The external network is said to be "in front of" the controller. An internal network connection is typically to the server-side of the network, which may include routers firewalls, caches, servers and other devices. The internal network is said to be

"behind" the controller. Network address translation normally occurs between the internal and external networks.

Any number of control segments may be coupled to the DFS over the message bus (see FIGURE 1). Typically, there is a primary CS and a secondary (or redundant) CS. Multiple control segments allow for load sharing between the control segments. In addition, fault tolerance is provided for since the primary CS can go out of service while the secondary CS assumes the role of the primary control segment. An example configuration with two control segments is shown in FIGURE 2.

As shown in FIGURE 2, a client (C1) is connected to an input port (1) of a data flow segment (DFS) through the Internet (external network). A first control segment (CS1) is connected to the DFS through ports 2 and 3 (INT and EXT). A second control segment (CS2) is connected to the DFS through ports 7 and 8 (INT and EXT). Content servers (N1, N2, N3) are connected to ports 4, 5 and 6. The content servers are behind the controller on the internal network.

A client communicates from the external network to the internal network through the DFS. The DFS communicates with the various control segments for instructions on new flows. The CS can access the networks that are connected to the DFS. The networks that are connected to the DFS may be of any type such as, for example, Fast Ethernet and Gigabit Ethernet. Administration and state sharing (if applicable) are provided over the messaging bus by either the internal or external interfaces (or both). Although the control segments are shown requiring two ports (EXT and INT), it is understood and appreciated that a single port connection will serve equally well.

The DFS categorizes packets into flows and performs translations on the packets in each flow. Translation is a set of rules that control which parts of a packet are to be rewritten, and the values that those parts will be rewritten to. Packets can be received by the DFS from both internal and external networks. After the packets are received, the DFS categorizes the packets into flows, analyzes the flow signature, and looks up the rules for that flow signature in a table (or another suitable data construct). If the table does not have an entry for the particular flow signature, the DFS sends a

query to the CS over the message bus for instructions. The CS replies with instructions on handling the new flow, and the DFS makes a new rule entry in the table for the new flow signature. The DFS routes, switches or otherwise directs the flow based on the rules for the particular flow signature. Thus, the DFS has capabilities that are similar to that of a router, a switch, or a routing-switch.

The DFS also detects certain events that occur for each flow. When an event that falls into a particular category (e.g. open a new connection) is detected, a message is sent from the DFS to the CS. The CS immediately responds with a message that describes translations and switching to perform on the flow (if required). The operation of the DFS will become apparent from the discussion that follows below.

A virtual server is an IP address and TCP/UDP port combination (the actual server is referred to as a "node"). The controller accepts a request from the virtual server and load balances those requests to servers situated behind the controller. Virtual servers are established by the CS and communicated to the DFS via the message bus. The DFS attempts to match the destination address and port number of packets received from the external network to a virtual server.

Overview of the operation of the DFS

FIGURE 3 shows a conceptual operation of an example scenario in which a client and a server exchange sequences of packets.

First, a client sends a first packet to the DFS. The DFS receives the packet and determines that the packet is not part of any flow currently being maintained. The DFS sends a message (QUERY) to the CS requesting instructions on how to handle the new flow (i.e which server shall packets be routed to). The CS receives the message from the DFS, determines how to handle FLOW A, and sends a message (REPLY) to the DFS with instructions on how to handle the flow. The DFS receives the message (REPLY) from the CS and stores the instruction in a local memory (table, etc.). Then, the DFS begins to process the packets for FLOW A and send the processed packets to the selected server.

A set of packets is sent from the server to the client in response to the server receiving FLOW A (i.e. a handshake, acknowledge, etc.). The server sends the packets to the DFS. The DFS receives the packets from the server and recognizes the packets as belonging to a return flow (FLOW B) of an existing communication by FLOW A. The DFS processes the packets and sends the processed packets to the selected client without intervention by the CS.

The client sends another set of packets to the DFS. The DFS receives the packets immediately recognizes that the packets belong to an existing flow (FLOW A). The DFS processes the packets and sends the processed packets to the selected server without intervention by the CS.

After receiving packets corresponding to FLOW A, the server responds by sending packets to the DFS. The DFS receives the packets from the server and recognizes the packets as belonging to an existing flow (FLOW B). The DFS processes the packets and sends the processed packets to the selected client without intervention by the CS.

The present invention may be implemented with varying wire protocols. Each wire protocol defines a communication protocol for message exchanges between the DFS and the CS (or multiple CSs). Although two different wire protocols are discussed below, any other suitable wire protocol is considered within the scope of the present invention.

SSMB Wire Protocol (Real Time Configuration Protocol)

In one example of the present invention, the messaging bus is structured as a state-sharing message bus (SSMB). The SSMB operates in real-time, has extremely low latency, and high bandwidth. Since the total latency of the controller includes the round-trip latency of the SSMB interface, it is important that the SSMB have low latency and a high bandwidth to provide adequate real-time operation. The SSMB bus structure is useful for systems where the DFS actively sends and receives messages to/from the CS (or multiple CS).

In one embodiment, the SSMB is layered on top of UDP. All message types fit into one UDP data-gram. Also, all message types should fit within on MAC frame to avoid IP fragmentation and reassembly.

The flow of messages from CS to DFS is asynchronous and independent of the flow of messages from the DFS to CS. Reply messages are initiated in response to query messages. The request and replies from the DFS and CS need not be interleaved. The querying segment should not be idle while waiting for a reply. The querying segment should not waste time trying to associate received replies with queries. Instead, reply messages should be self-contained so that the receiving segment will process the reply without needing to know what the original query was.

Each message contains a serial number (or other indicator) that is generated by the originator of the flow. The originator of the flow is the first party to query or notify the other party regarding the flow. The serial number remains constant for all messages pertaining to the flow during the flow's lifetime. Since the DFS is typically the first party to detect an inbound flow (from the external network to the internal network), the DFS is generally the originator of the flow. In this case, the DFS sends a message (QUERY) to the CS to notify the CS about the new flow. In some instances (e.g. CS-assisted flow), the CS originates the flow by sending a message (NEWFLOW) to the DFS.

In one example of the present invention, message types are defined as depicted in table I shown in FIGURE 4. A "Y" entry in the "DFS Sends?" column indicates that the message is sent from the DFS to the CS (or multiple CSs). A "Y" in the "CS Sends?" column indicates that the message is sent from the CS to DFS. An "H" priority indicates a time critical message, usually because the latency of packets in a flow is proportional to the time for the message and it's reply. An "L" priority indicates that the message is not a time-critical message. A "Y" in the "Batching?" column indicates that the information portion of the message may be combined with other messages such that the messages are sent as a batch. A "Y" in the "Single mbuf?" column indicates that the message is required to fit in a single memory buffer of the DFS or CS as is applicable. A "Y" in the "State Sharing" column indicates that the

message is to be replicated to the standby CS (or multiple CSs) if there is one. The "Expected Response" column indicates the type of message that is expected in response to the current message.

FIGURE 5 shows a table (table II) listing the data elements that are expected to be sent in a given message. Each message type may consist of a predefined subset of these data elements. The length of the message is derived from the UDP header.

FIGURE 6 shows a table (table III) of message fields for the message types defined in FIGURE 4. After having read the present disclosure, it is understood and appreciated that other message types and message fields may also be utilized within the scope of this invention. The message layout is optimized for efficient processing by the CS and according to the needs of the particular DFS. Every message has a message header that includes msg_type, error_code, and message serial number fields. Example message layouts are shown in FIGURE 9.

FIGURE 7 shows a table of exemplary boolean variables that are packed into the flags field shown in FIGURE 6. The OUTBOUND variable determines if a message concerns an inbound flow or an outbound flow (TRUE=outbound, FALSE=inbound). The majority of messages are regarding inbound flows. The ADD_TCP_OFFSETS variable determines if the TCP packet is to be offset or not (TRUE=offset TCP packet, FALSE=do not offset TCP packet). When an offset is to be added to the TCP packet, the seq_offset and ack_offset variables are used to determine the amount of offset to be added to the values found in the TCP header. TCP packets are often offset when the TCP handshake proxy is performed by the CS.

FIGURE 8 shows a table of exemplary error codes that are used in the error field shown in FIGURE 6. A code of UNKNOWN indicates that the CS does not know how to handle a particular flow. A code of NOTAVAIL indicates that the virtual server that was requested in a QUERY message by the DFS is not available because either the port requested was denied or the virtual server is in maintenance mode. A CONNLIMIT error code indicates that the requested connection (in a QUERY from the DFS) would exceed the maximum available connections to the virtual server. A

CONNALIVE error code indicating that a flow subject to a SSMB REAP message (requesting deletion of a flow) from the CS is still active. The DFS sends a STATS message with this error field set to request the CS to increase the statistics on the flow.

When a redundant CS topology is used, one CS is the primary (active) controller, and the remaining CS (or multiple CSs) is a backup or standby controller. The standby CS receives copies of all state information originating in the active CS. Specifically, the standby CS needs to receive a copy of any state information message that is communicated by the active CS to the DFS so that all of the CSs share common state information. Exemplary state information messages are shown in FIGURE 6.

The DFS is designed to facilitate shared state information across multiple control segments (CSs) using any appropriate communication method including, but not limited to, IP multicasting and artificial packet replication.

An IP multicast message joins all control segments (CSs) and the DFS into a common IP multicast group. The active CS sends all state information messages as multicast messages. The DFS receives the multicast message, replicates the message, and sends the replicated message to all members of the IP multicast group. All other non-multicast messages are pointcast on a separate non-multicast address.

For certain DFS implementations, an artificial packet replication method is simpler to implement than IP multicast messaging. The effect of artificial packet replication is the same as multicast in that the DFS creates a replica of the received state information and forwards a copy of the state information to the standby control segment(s). However, the active CS is not required to send the information to the DFS as a multicast message as in the IP multicast message method.

The CS will not respond to an SSMB message unless the DFS has correctly responded to an authentication challenge. The format for an authentication message is shown in FIGURE 6 and FIGURE 9. Authentication will be discussed in further detail as follows below in this specification.

CSMB Wire Protocol

Another type of messaging bus structure is a configuration sharing message bus (CSMB). The CSMB works in concert with the SSMB Wire Protocol. The CSMB does not need to operate in real-time. In one example, the CSMB is layered on top of TCP. The protocol is carried by one or both of the network connections between the CS and the DFS.

The DFS passively monitors the message bus. The CS actively makes connections, while the DFS accepts connections. The CS automatically connects whenever it detects that it is not already connected. The DFS supports simultaneous connections between the DFS and multiple control segments (CSs).

Whenever a new connection is established the CS (or multiple CSs) and the DFS send HELLO messages (e.g. see FIGURE 10) to one another prior to any other message types. The CS will also periodically send a HELLO message to the DFS to validate the connection. The CS configures the time interval between the transmissions of HELLO messages.

The CSMB wire protocol provides for an asynchronous flow of messages from the CS (or multiple CSs) to the DFS. The flow of messages from the CS to the DFS is independent of the flow of messages from DFS to CS. Reply messages are initiated in response to query messages. The requests and replies from the DFS and CS need not be interleaved. The querying segment should not be idle while waiting for a reply. The querying segment does not waste time trying to associate received replies with queries. Instead, reply messages are self-contained so that the receiving segment will process the reply without needing to know what the original query was.

Each message contains a serial number (or other indicator) that is global for all messages, and a serial number that is specific to messages of the specific type. In one embodiment the serial numbers are unsigned 16-bit integers.

According to one embodiment of the invention, message types are defined in a table (table IV) as shown in FIGURE 10. A "Y" entry in the "DFS Sends?" column indicates that the message is sent from the DFS to the CS (or multiple CS). A "Y" in the "CS Sends?" column indicates that the message is sent from the CS to DFS.

The "Expected Response" column indicates the type of message that is expected in response to the current message.

FIGURE 11 is a table (table V) listing data elements that are expected to be sent in a given message. Each message type may consist of a predefined subset of these data elements. The length of the message is derived from the UDP header.

FIGURE 12 shows a table (table VI) of message fields for the message types defined in FIGURE 10. It is understood and appreciated that other message types and message fields may also be utilized within the scope of this invention. The message layout is optimized for efficient processing by the CS and according to the needs of the particular DFS. Every message has a message header that includes msg_type, serial_global, serial_bytype and msg_length fields. Example message layouts are shown in FIGURE 14.

Version number fields (vers_major, vers_minor) apply to both CSMB and SSMB wire protocols. In the HELLO messages (see FIGURES 12 and 14), the CS and DFS attempt to negotiate the highest numbered version that is supported by both.

Two different virtual server classes are supported, a CS-assisted virtual server and a DFS-assisted virtual server (see FIGURE 13). Virtual servers that do not require application data load balancing are of the class DFS_ASSIST. The DFS-assisted virtual server has no special flag settings in the ADD_VS and VS_LIST messages.

As discussed previously, virtual servers (defined as an IP address and TCP/UDP port combination) are established by the CS and communicated to the DFS via the message bus. For CSMB wire protocol, the CS is configured to automatically inform the DFS of each deletion and addition of virtual servers.

The controller accepts a request from the virtual server (sometimes referred to as a "node") and load balances those requests to servers situated behind the controller. The DFS attempts to match the destination address and port number of packets received from the external network to a virtual server.

In one embodiment of the invention, the DFS performs TCP handshake proxy (also referred to as TCP splicing) for certain types of virtual servers, and also

extracts application specific data from the client request. The DFS sends a message to the CS that includes the extracted data (SSMB APP_QUERY, see FIGURES 6 and 9). Virtual servers that are supported according to this embodiment are of the class DFS_ASSIST. These virtual servers are specified using the ADD_VS and VS_LIST messages (see FIGURES 10, 12 and 14). Setting the flags for SSL_PROXY, HTTP_PROXY, specifies the DFS_ASSIST type virtual servers or COOKIE_PROXY, as will be discussed later.

In another embodiment of the invention, the DFS does not have the capability to extract application data from the client request because it is an unsupported application protocol. In this instance, the DFS will perform the TCP handshake proxy with the client and forward (bridge) a copy of the client packets to the CS (or multiple CSs). Virtual servers that are supported according to this embodiment are of the class DFS_ASSIST. These virtual servers are specified using the ADD_VS and VS_LIST messages (see FIGURES 10, 12 and 14). Setting the flags field to RAW_PROXY (discussed later) specifies the DFS_ASSIST type for unsupported application protocols.

Virtual servers may also be of the class CS_ASSIST, where the CS routes all packets in a flow that are related to a specific CS assisted virtual server. In this instance, the DFS bridges all packets to and from the CS, and state or configuration data is not exchanged between the DFS and the CS. The CS_ASSIST class of virtual servers is used when the DFS is incapable of performing the TCP handshake proxy. The CS_ASSIST class of virtual servers is also used when the DFS is incapable of assuming a flow from the CS using a hybrid CS assisted virtual server.

A hybrid CS assisted virtual server is used in conjunction with the TCP handshake proxy. Flows begin as CS_ASSIST and then switch over to DFS_ASSIST after the TCP handshake proxy is complete and a message (NEWFLOW) is received from the CS. For TCP flows, the DFS adds the sequence number and ack offsets received in the NEWFLOW message to all the packets received in the flow.

FIGURE 15 shows a table of exemplary Boolean variables that are packed into the flags field shown in FIGURE 12. The TRANSLATE_ADDR variable determines if the DFS will provide address translation functions for an incoming flow

(TRUE=translate, FALSE=do not translate). The TRANSLATE_PORT variable determines if the DFS will provide port translation functions for an incoming flow (TRUE=translate, FALSE=do not translate). The ROUTE_BY_DST_IP variable determines if the DFS will perform a route address lookup (TRUE) or if the DFS will use the next_hop_ipaddr field to determine the next hop for the flow (FALSE).

The REDUNDANT and APP_PROXY variables are part of the HELLO message. When REDUNDANT is set to TRUE, multiple CSs are used and the ssmb_standby_ipaddr is used for state sharing. When state sharing is not used or multiple CSs are not used, the REDUNDANT variable is set to FALSE. When the WILDCARD_ADDR variable is set to TRUE, the virt_ipaddr field is ignored and all traffic received from the external network that is destined for any address that does not match another virtual server or other known local address is processed as if it was addressed to this virtual server. When the WILDCARD_PORT variable is set to TRUE, the virt_port field is ignored and all traffic received that is destined for any virtual port that does not match the virtual port of any other virtual server is processed as if it was addressed to this virtual server. If the NOARP_MODE is set to TRUE then the controller acts like a router and accepts packets destined to the address but does not respond to ARP requests for the address. When NOARP_MODE is set to FALSE, the controller acts as a host and advertises the address (e.g., responds to ARP requests). When APP_PROXY is set to TRUE, the controller supports application data load balancing, and can perform the TCP handshake proxy as well as extract application data from the client request. The CS typically sets APP_PROXY to TRUE. The DFS set APP_PROXY to TRUE if the DFS has sufficient capability to do so. If SSL_PROXY is set to TRUE then the CS makes load-balancing decision for the virtual server based upon the client's SSL session id. The DFS proxies the client connection, extracts the session id, and sends the session id to the CD. If COOKIE_PROXY is set to TRUE then the CS makes load-balancing decisions for the virtual server based upon the value of a cookie in the HTTP request. The DFS proxies the client connection, extracts the designated cookie, and send the cookie to the CS with the cookie name provided in the app_data field. If HTTP_PROXY is set to TRUE then the CS makes load-balancing

decisions for the virtual server based upon the value of the HTTP request. The DFS proxies the client connection, extracts the HTTP request, and sends the data to the CS. If RAW_PROXY is set to TRUE then the CS makes load-balancing decisions based upon an application data format that is not supported by the DFS. The DFS proxies the client connection and bridges packets that are received from the client to the CS.

FIGURE 16 shows a table of exemplary error codes that are used in the error field shown in FIGURE 12. A code of VERS_NEW indicates that the particular segment does not yet support the version specified in the HELLO message. If possible, the other segment should send another HELLO message with a lower version number. A code of VERS_OBSOLETE indicates that the particular segment no longer supports the version specified in the HELLO message. In this case, the other segment should send another HELLO message with a higher version number.

In one embodiment of the invention, CSMB messages are used to negotiate IP addresses and UDP port numbers for the SSMB wire protocol. CSMB may also be used to exchange configuration information such as the default gateway address and other administrative IP addresses. CSMB is also used to negotiate versions between the CS and DFS for both the SSMB and CSMB wire protocols.

The CS sends authentication challenges to the DFS using the CSMB wire protocol. The CS also sends other messages to the DFS such as the addition or deletion of each virtual server. The CS will not respond to a CSMB message from the DFS unless the DFS has correctly responded to an authentication challenge. The format for an authentication message is shown in FIGURE 12 and FIGURE 9. Authentication will be discussed in further detail as follows below in this specification.

Operation of the DFS in SSMB mode

FIGURE 17 shows a flow chart of the basic operation of the DFS. Processing begins at start block 1710 and proceeds to block 1720 where the DFS waits for the receipt of a new packet. When a new packet is received, processing proceeds to block 1730. Proceeding to block 1740, the DFS analyzes the new packet to determine if the packet is part of a new flow.

When the incoming packet is part of an existing flow, processing proceeds to block 1770 where the incoming packet is processed and subsequently sent to the selected client or server as may be required. Processing then proceeds to block 1720 where the DFS waits for the receipt of another packet.

Returning to decision block 1740, when the incoming packet is identified as part of a new flow, processing proceeds to block 1750 where the DFS sends a message (QUERY) to the CS for instructions on handling the new flow. Processing then proceeds from block 1750 to block 1760 where the DFS receives an instruction from the CS (REPLY) on how to handle the new flow. The DFS then stores the instruction from the CS in a local memory area (i.e. a table). Once the DFS has stored the processing instructions, processing proceeds to block 1770 where incoming packets are processed and the packets are sent to the selected client or server (as is required).

Although the processing is shown as sequential in FIGURE 17, the DFS continually receives packets and messages from the CS in an asynchronous manner. Since the DFS is continually receiving packets and messages, the DFS may not enter an idle state and continues processing messages and packets from a local memory buffer.

Flow Management

As discussed previously flows have a finite lifetime. The DFS and CS have different notions concerning when an existing flow ends. The DFS and CS have independent creation and deletion of flows.

An overview of the operation of the DFS procedure that is used to determine if a flow is still alive is shown in FIGURE 18. Processing begins at start block 1810 and proceeds to decision block 1820 where the DFS determines if a TCP shutdown (FIN or RST) is detected within the flow. When a TCP shutdown is detected within the flow, processing proceeds to block 1860 where the flow is deleted.

Returning to decision block 1820, when the DFS determines that the flow does not contain a TCP shutdown, processing proceeds to decision block 1830 where the DFS determines if an overflow has occurred in a flow table (DFS flow table). As discussed previously, the DFS maintains a table that keeps track of each flow

signature and rules for processing each flow. When the table no longer has enough room to accommodate additional entries, an overflow condition occurs. When the overflow condition has occurred, processing proceeds from decision block 1830 to block 1860 where the flow is deleted.

Returning to decision block 1830, when the DFS determines that the flow table has not overflowed, processing proceeds to decision block 1840 where the DFS determines if a flow timeout has occurred. When the timeout condition has occurred, processing proceeds from decision block 1840 to block 1860 where the flow is deleted.

Returning to decision block 1840, when the DFS determines that the flow timeout has not occurred and processing proceeds to block 1850 where the DFS determines that the flow is still alive. From block 1850, processing proceeds to block 1870 where the flow management for the DFS is complete. In one embodiment of the invention, the DFS sends a message to the CS to inform the CS that the flow is still active. In another embodiment of the invention, all messages that are sent from a particular CS are monitored by all CSs so that each CS may maintain duplicate table entries for fault tolerance.

As described above, the DFS deletes flows when: a normal TCP shutdown (FIN or RST) is detected within the flow, an overflow in the flow table occurs, or a timeout for the flow has expired. The DFS also deletes flows when the CS has sent a REAP or TIMEOUT message about the flow and the DFS does not consider the flow to be active.

An overview of the operation of the CS procedure that is used to determine if a flow is still alive is shown in FIGURE 19. Processing begins at start block 1910 and proceeds to decision block 1920 where the CS determines if a TCP shutdown (FIN or RST) is detected within the flow. When a TCP shutdown is detected within the flow, processing proceeds to block 1950 where the flow is deleted.

Returning to decision block 1920, when the CS determines that the flow does not contain a TCP shutdown and processing proceeds to decision block 1930 where the DFS determines if a flow timeout has occurred. When the timeout condition

has occurred, processing proceeds from decision block 1930 to block 1950 where the flow is deleted. Otherwise, if no flow timeout has occurred, processing proceeds from decision block 1930 to block 1940 where the CS determines that the flow is active.

In one embodiment of the present invention, the CS sends a message to the DFS (e.g. REAP) when the CS has determined that a flow should be terminated due to inactivity. If the DFS determines that the flow is still active, the DFS sends a message (e.g. STATS) to the CS that includes an error code indicating the flow is still alive (e.g. CONNALIVE). When the CS receives the STATS message indicating a flow is active, the CS will not delete the flow and will reset the inactivity timer for the flow.

As discussed previously, the CS performs high-level control functions such as load-balancing based upon various statistics that are gathered by the controller. The CS also keeps track of statistics for each flow and determines when a particular flow has become inactive. A timer is associated with each flow and may be used to determine factors such as: most active flow, least active flow, time flow opened, most recent activity as well as other parameters related to flow statistics and load balancing.

When a flow is detected as timed out, the CS sends a message to the DFS to delete the flow (e.g. REAP). The DFS sends a message that is responsive to the REAP indicating that either the flow is still active (e.g. STATS message with CONNALIVE set to true) or that the flow has been deleted. The CS maintains the flow as active while waiting for the response from the DFS. The CS will either delete the flow from the CS flow tables or reset the timer for the flow after receiving the message from the DFS.

A flow in the DFS may end before a TCP session ends. Consequently, the DFS may start a new flow for an existing TCP session and query the CS about it. The CS will be able to distinguish between a new DFS flow and a new TCP session by examining messages that are received from the DFS.

A new flow is detected when an event is detected by the DFS. Events that are detected by the DFS are communicated to the CS (or multiple CSs) via the

message bus (e.g. SSMB). In one example, a TCP or UDP connection-open is detected by the DFS indicating the start of a new flow.

There are several different types connections that can be opened using TCP. Each type of connection results in an event that is detected by the DFS, and requires a different response by the DFS. UDP flows are handled similar to TCP events. The various connection open/close types will be discussed as follows below.

TCP Connection Open, Non-Application Proxy DFS-Assisted Virtual Servers

When a TCP connection open is detected of this type, the DFS sends a message (QUERRY) to the CS that contains the source and destination IP addresses and port numbers. While the DFS awaits a reply from the CS, the DFS buffers all incoming packets in the flow (identified by its flow signature as described previously) until it receives instructions from the CS. The CS will respond to the QUERRY with a message (REPLY) describing translations to make for packets in that flow, and information that will be used to switch the flow.

TCP Connection Open, Application Proxy Virtual Servers

When a TCP connection open is detected of this type, the DFS performs a handshake with the client before proceeding. After a TCP handshake proxy is made with the client, the DFS buffers packets received from the client until the required data from the request is received. The DFS sends a message (e.g. APP_QUERY) to the CS that contains the source and destination IP addresses and ports as well as the application data from the client request. The CS will respond to the APP_QUERY with a REPLY message describing the translations to make for packets in that flow, and information that will be used to switch the flow. Once the REPLY is received from the CS, the DFS performs a handshake proxy with the server and establishes an outbound flow. The DFS continues to buffer all packets in the flow until the TCP handshake proxy with the server is completed, and the flow established.

TCP Connection Open, Raw Proxy Virtual Servers

When a TCP connection open is detected of this type, the DFS performs a handshake with the client before proceeding. After a TCP handshake proxy is made with the client, the DFS forwards (bridges) packets received from the client to the CS until the required amount of payload data is received. When the CS has received sufficient data to make a load balancing decision, the CS will send a message (NEWFLOW) to the DFS describing the translations to make for the packets in the flow. Once the DFS receives the NEWFLOW message, the DFS performs the TCP handshake proxy with the server and establishes the outbound flow. The DFS continues to buffer all packets in the flow until the TCP handshake proxy with the server is completed, and the flow established.

TCP Connection Open, Hybrid CS-Assisted Virtual Servers

When a TCP connection open is detected of this type, the CS performs a handshake with the client and receives data from the client. The CS continues to receive data from the client until sufficient data is received to make a load balancing decision. When the CS has made a load balancing decision, the CS will perform a TCP handshake with the server and send a message (NEWFLOW) to the DFS describing the translations to make for the packets in the flow. Once the DFS receives the NEWFLOW message, the DFS will assume the flow, applying the TCP sequence number and offsets received in the NEWFLOW message, to the continuing packets in the flow.

TCP Connection Close

The DFS keeps track of the TCP connection termination protocol for the flow (Application FIN, Server ACK, Server FIN, and Application ACK). The DFS should not delete a TCP flow when the flow is in the middle of state transition. When a TCP connection close (or reset) is detected, the DFS notifies the CS by sending a message (DELETE) to the CS. The DFS does not need to wait for a response from the CS and may stop tracking the flow. In a system that includes statistics gathering

mechanisms in the DFS, the DFS will include the statistics for the flow in the DELETE message.

DFS Message Processing

Messages received by the DFS from the CS are generally described as shown in FIGURE 20. Processing begins at start block 2010 and proceeds to block 2020 where the DFS begins parsing the received message. When the message (e.g. REAP FLOW) from the CS is to delete a particular flow (as specified in the message), processing proceeds to block 2025, where the DFS updates the DFS tables based on flow activity (refer to the previous discussion of FIGURE 18). After the DFS flow activity is updated (either the flow is still alive, or deleted from the DFS table), processing is complete and processing ends at block 2080.

Returning to block 2020, processing proceeds to block 2030 when the received message is a message other than deleting a particular flow. At decision block 2030, the DFS determines if the received message indicates a particular flow has timed out. When the message received from the CS indicates that the particular flow has timed out, processing proceeds to block 2025, where the DFS updates the DFS flow tables based upon the activity of the particular flow. Processing proceeds from block 2030 to block 2040 when the message received from the CS indicates that the message is not a flow timeout.

At decision block 2040, the DFS determines if the received message indicates a new flow is to be set up. When the message indicates a new flow, processing proceeds to block 2045 where the new flow is processed. As discussed previously, new flows are processed in accordance with routing/switching instructions that are provided by the CS, and entered into the DFS flow tables accordingly. When the message is parsed as a message other than a new flow, processing proceeds from block 2040 to block 2050.

At decision block 2050, the DFS determines if the received message indicates to reset the system. When the system is to be reset, processing proceeds from decision block 2050 to block 2055 where all flows are deleted from the DFS flow

tables. When the message does not indicate a system reset, processing proceeds from block 2050 to block 2060.

At block 2060, the DFS determines that an invalid message has been received. Alternatively, the DFS may process other message types as may be required in a particular application. From block 2060, processing proceeds to block 2070 where processing is completed. Processing also proceeds to block 2070 from blocks 2025, 2045, and 2055.

One example of new flow processing (e.g. 2045 in FIGURE 20) is shown in FIGURE 21. Processing begins at start block 2110 and proceeds to block 2120 where the DFS begins to parse the new flow instruction from the received CS message. The new flow has a corresponding flow signature, destination, source as well as any other relevant routing or switching information that is described by the CS in the received message. After the DFS extracts the relevant information from the received message, processing proceeds from block 2120 to block 2130.

At block 2130, the DFS determines if the new flow signature corresponds to an existing flow entry in the DFS tables. When the new flow signature is part of an existing flow entry, processing proceeds from block 2130 to block 2170 where the DFS flow table is updated. When the new flow signature is not recognized by the DFS (not in the DFS flow table), processing proceeds from decision block 2130 to decision block 2140.

At decision block 2140, the DFS determines if the DFS flow table is full. When the DFS flow table is full, processing proceeds to block 2150 where the DFS creates a free space in the DFS flow table. The space may be made available by any appropriate criteria such as, for example, deleting the oldest flow signature entry. Processing proceeds from block 2150 to block 2160 where the new flow signature is entered into the DFS flow table based upon the instructions provided by the CS. Processing also proceeds to block 2160 from decision block 2140 when the DFS table is not full. From block 2160, processing proceeds to block 2170 where processing is concluded.

CS Message Processing

Messages received by the CS from the DFS are generally described as shown in FIGURE 22. Processing begins at start block 2210 and proceeds to block 2220 where the CS begins parsing the received message. At block 2220 the CS determines if the received message is a request for instructions (e.g. QUERY) related to a particular flow. When the message indicates a request for instructions, processing proceeds from block 2220 to block 2225. Otherwise processing proceeds from block 2220 to block 2230. At block 2225, the CS analyzes the particular flow described in the received message and determines how to the DFS is to handle routing or switching the packets related to that flow. The CS may use any appropriate method to determine the routing/switching of packets such as, for example, based upon load balancing. The CS subsequently sends a message (e.g. REPLY) to the DFS containing instructions for handling the flow.

At block 2230 the CS determines if the received message is an application request for data from a server. When the message indicates such a request (e.g. AP_QUERY), processing proceeds from block 2230 to block 2225. Otherwise processing proceeds from block 2230 to block 2240.

At block 2240 the CS determines if the received message is a request from the DFS for a particular flow to be deleted from the CS flow tables. When the message indicates that the particular flow is to be deleted (e.g. DELETE), processing proceeds from block 2240 to block 2245. Otherwise, processing proceeds from block 2240 to block 2250. At block 2245, the CS deletes the particular flow from the CS flow table.

At block 2250, the CS determines if the received message is a request from the DFS (or another CS) to reset the system. When the message indicates a system reset (e.g. RESET), processing proceeds from block 2250 to block 2255. Otherwise, processing proceeds from block 2250 to block 2260.

At block 2260, the CS determines that an invalid message has been received. Alternatively, the CS may process other message types as may be required in a particular application. From block 2260, processing proceeds to block 2270 where

processing is completed. Processing also proceeds to block 2270 from blocks 2225, 2245, and 2255.

The logical operations of the various embodiments of the invention are implemented as a sequence of computer implemented actions or program modules running on one or more computing devices; and/or as interconnected hardware or logic modules within the one or more computing devices. The implementation is a matter of choice dependent on the performance requirements of the computing system implementing the invention. Accordingly, the logical operations making up the embodiments of the invention described herein are referred to alternatively as operations, actions or modules. Program modules may be described as any construct (e.g. routines, programs, objects, components, and data structures) that perform particular tasks or implement particular abstract data types. The functionality of program modules may be combined or distributed in various embodiments.

The above specification, examples and data provide a complete description of the manufacture and use of the composition of the invention. Since many embodiments of the invention can be made without departing from the spirit and scope of the invention, the invention resides in the claims hereinafter appended.